Table of Contents

NodeT (Class) 3

ForestT (Class) 5

[Advance (Class) 2](#__RefHeading___Toc2189_279954108)

# Advance (Class)

Properties:

A name

A list of requirements.

Behaviors:

Build an advance from a string

Return the advance’s name

Return the advance’s list of requirements.

Discussion:

The input string to the constructor consists of

+ A name

+ a comma

+ A list of requirements for this advance.

+ a single dash for no requirements.

+ a semicolon separated list of strings for requirements.

Examples

Pottery,-

Politics,Monarcy

Geography,Mathematics;Sailing

**Class:Advance**

**Method: Constructor**

Narrative: Build an advance.

Input: string

Output: none

*parse the input line to find the name and requirements.*

**Method**: **Name**

Narrative: return the name of the advance

Input: none

Output: a string

**Method: Requirements**

Narrative: return the list of strings that name requirements for this advance.

Input: none

Output: a vector of strings

# NodeT (Class)

This class will be the container for elements of a graph. This is a support class for the ForestT. This class should be implemented in Node.h and Node.C/cpp

Properties:

An advance.

A list of pointers to children of this node.

A list of pointers to parents of this node.

Behaviors:

Return the data stored in the node.

Add a parent or child pointer.

Return a vector of parent pointers.

Return a vector of child pointers

Check to see if this node is the parent of a given advance

Check to see if this node is the child of a given advance.

Discussion:

There should be no default constructor, copy constructor, or assignment operator. A NodeT should not be copied as the child and parent pointer lists will almost certainly be wrong.

There is no destructor necessary, as this class is NOT responsible for managing the pointers it stores. These pointers are for reference only.

**Class:NodeT**

**Method: Constructor**

Narrative: Build an nodeT

Input: AdvanceT

**Method**: **AddParent**

Narrative: Adds the given node as a parent of the node, but only if it is not null and not already a parent. It should also add the it’s self as the child of the given node.

Input: a node pointer

Output: none

Algorithm:

*If the input node is not null*

*if the input node is not a parent of this node*

*add the input node as a parent of this node.*

*Add this node as a child of the input node.*

**Method: AddChild**

Narrative: Check to see if the given node is a child of this node, if not add it. Add this node as a parent of the given node.

Input: A node pointer

Output: none

*if the input node is not null*

*if the input node is not a child of this node*

*add the input node as a child of this node*

*add this node as a parent of the input node*

**Method: Parents**

Narrative: return a list of pointers to the parents of this node.

Input: none

Output: a constant vector of node pointers

**Method: Children**

Narrative: return a list of pointers to the children of this node.

Input: none

Output: a constant vector of node pointers

**Method: Advance**

Narrative: return the data stored in this node.

Input: none

Output: an AdvanceT

**Method: IsParent**

Narrative: Return true if this node is a parent of the input advance

Input: an Advance

Output: a boolean

*if the input advance matches the advance of any children of this advance return true.*

**Method: IsChild**

Narrative: Check to see if this node is the child of the input advance.

Input: an advance

Output: A boolean

*if the input advance matches the advance of any of the parents of this advance return true*

# ForestT (Class)

This class contains a forest of advances. It is constructed of NodeT elements. This class should be constructed in Forest.h and Forest.C/cpp

Properties:

A vector of NodeT elements.

Behaviors:

Construct a forest of advance trees.

Check to see if an advance exists

Print all advance trees from the parent of the tree in a forest

Print a tree representing the parents of an advance

Print a tree representing the children of an advance

NOTE:

A parent is a base advance that requires to others to obtained

A child is an advance that requires a base advance in order to be obtained.

Discussion:

This class relies on the Depth First Search Algorithm for several methods.

*DFS(NodeT \* node)*

*if the node is null return*

*perform an action at the node*

*for each child node*

*DFS(child node)*

**Class: ForestT**

**Method: Default Constructor**

Narrative: This constructor should call the single parameter string constructor with a file name of “Advance”

Input: none

Output: none

**Method: Single string parameter constructor**

Narrative: This constructor should open the named by the input parameter.

Input: A string representing the file name.

Output:

*Open the file.*

*While Input*

*AddAdvance(AdvanceT(line))*

*FixAdvances()*

**Method: AddAdvance**

Narrative: This method is private. It adds an advance to the advance forest

Input: an advance

Output: none

*If the advance is in the forest throw ADVANCE\_ALREADY\_EXISTS*

*For each requirement of the of the advance*

*if the requirement is in the forrest add the advance as a child of the requirement.*

*If the node was not added as a child node*

*add the node to the vector of nodes in the forest.*

**Method: FindAdvance**

Narrative: Return a pointer to a node that matches the advance name or the null pointer. This is a private method.

Input: an advance name.

Output: A NodeT pointers

*for each node in the list of trees*

*if the tree contains the named advanced (Do a DFS search on the tree)*

*return a pointer to the node containing the named advance*

*return null*

**Method: DFS**

Narrative: Search a tree for a named node. This is a private method.

Input: A pointer to a tree or subtree

a string naming a node

Output: A pointer to a node with the named attribute or the null pointer.

*Conduct Depth First Search on the given tree.*

*Return a pointer if the named advance is in the tree*

*return null if it is not.*

**Method: FixAdvances**

Narrative: Remove all non basic nodes from the forest list and add them to the appropriate graphs. This is a privet method.

Input: none

Output: none

*for each node in the forest*

*if it has prerequisites*

*remove it from the forrest*

*find each prerequisite node*

*add the given node as a child to the prerequisite node*

*for each node in the forest*

*FixParents(node)*

**Method: FixParents**

Narrative: Fix the parents (and the children) on all nodes in the tree. This is a private method. This uses the DFS algorithm.

Input: a node pointer to a tree or sub tree.

Output:

*for all requirements for this advanced*

*find the parent node representing the requirements*

*if the node is not a child of the parent*

*add this node as a child to the parent.*

*Fix Parents for all the children of this node*

**Method: Destructor**

Narrative: Delete all memory used by this class

Input:

Output:

*create an empty vector*

*call KillTree on each node in the forest*

*Delete each node in the vector just constructed*

**Method: KillTree**

Narrative: Remove all nodes stored in a tree. This is a private method. This method employs the DFS algorithm.

Input: A node pointer

A vector of node pointers.

Output: none

*If the input node pointer is not in the vector of node pointers add it to the vector.*

*Call KillTree on each child of the input pointer.*

**Method: AdvanceExists**

Narrative: Return true if the named advance is in the forest.

Input: a string

Output: boolean

*for each tree in the forest*

*if (DFS(name) != null*

*return true*

*return false*

**Method: PrintParents**

Narrative: Print the parents of the named advance.

Input: a string

Output:

*Find the named attribute*

*Call DFSPrint on the node, depth 0, going up*

**Method: PrintChldren**

Narrative: Print the children of the named advance

Input:

Output:

*Find the named attribute*

*Call DFSPrint on the node, depth 0, going down*

**Method: PrintAll**

Narrative: print all trees in the forest

Input:

Output:

*Call PrintChildren on each tree in the forest.*

**Method: DFSPrint**

Narrative: Use the DFS algorithm to print a tree.

Input: a node pointer to the tree or subtree,

a depth

the direction to print

Output:

*print depth tabs*

*print the name of the attribute at this node followed by a new line*

*if the direction is up call DFSPrint on all parents, depth = depth+1, direction = up*

*else call DFSPrint on all children, depth = depth+1, direction = down*